DOCUMENT RESUME

ED 232 861 SE 042 635

AUTHOR Novak, Gordon S., Jr.
TITLE Data Abstraction in GLISP.
INSTITUTION Stanford Univ., Calif. Dept. of Computer Science.
SPONS AGENCY Advanced Research Projects Agency (DOD), Washington, D.C.; National Science Foundation, Washington, D.C.
PUB DATE [82]
CONTRACT MDA-903-80-c-007
GRANT SED-7912803
NOTE 11p.; For related documents, see SE 042 630-634.
PUB TYPE Reports - General (140)

EDRS PRICE MF01/PC01 Plus Postage.
DESCRIPTORS *Computer Programs; *Computer Science; *Programing; *Programing Languages
IDENTIFIERS *Data Abstraction; *GLISP Programing Language; National Science Foundation

ABSTRACT

GLISP is a high-level computer language (based on Lisp and including Lisp as a sublanguage) which is compiled into Lisp. GLISP programs are compiled relative to a knowledge base of object descriptions, a form of abstract datatypes. A primary goal of the use of abstract datatypes in GLISP is to allow program code to be written in terms of objects, but independent of the actual implementation of objects; this allows the same code to be effective for objects which are implemented in different ways. An integrated programming environment is provided by the language, which includes editors for programs and datatype descriptions, interpretive programming features, and a display-based inspector/editor for data. GLISP program syntax includes PASCAL-like control structures and infix arithmetic expressions. In addition, the GLISP language is easily extendable. Operator overloading for user-defined objects occurs when arithmetic operators are defined as messages for those objects. The compiler can compile optimized code for access to objects represented in user-specified "representation languages," including database systems; GLISP has thus far been interfaced to the representation languages GIRL and LOOPS. GLISP is fully implemented and is available for six different dialects of LISP. (JN)

****
* Reproductions supplied by EDRS are the best that can be made *
* from the original document. *
****

ED232861

U.S. DEPARTMENT OF EDUCATION
NATIONAL INSTITUTE OF EDUCATION
EDUCATIONAL RESOURCES INFORMATION CENTER (ERIC)
✓ This document has been reproduced as received from the person or organization originating it.
☐ Minor changes have been made to improve reproduction quality.
• Points of view or opinions stated in this document do not necessarily represent official NIE position or policy.

Data Abstraction in GLISP[1]

Gordon S. Novak Jr.
Heuristic Programming Project
Computer Science Department
Stanford University
Stanford, CA 94305

"PERMISSION TO REPRODUCE THIS MATERIAL HAS BEEN GRANTED BY
National Science Foundation
TO THE EDUCATIONAL RESOURCES INFORMATION CENTER (ERIC)."

Abstract

GLISP is a high-level language which is based on Lisp and is compiled into Lisp. It provides a versatile abstract datatype facility with hierarchical inheritance of properties and object-centered programming. The object code produced by GLISP is optimized, so that it is about as efficient as hand-written Lisp. An integrated programming environment is provided, including editors for programs and datatype descriptions, interpretive programming features, and a display-based inspector/editor for data. GLISP is fully implemented.

1. Introduction

GLISP [8] [9] [10] is a high-level language, based on Lisp and including Lisp as a sublanguage, which is compiled into Lisp[2]. GLISP programs are compiled relative to a knowledge base of object descriptions, a form of abstract datatypes [13] [6]. A primary goal of the use of abstract datatypes in GLISP is to allow program code to be written in terms of objects, but independent of the actual implementation of objects; this allows the same code to be effective for objects which are implemented in different ways. A corollary goal is that the code generated for access to objects should be about as efficient as equivalent code written in the underlying base language, Lisp. GLISP is fully implemented, and is available from the author for six different dialects of Lisp.

GLISP program syntax includes PASCAL-like control structures and infix arithmetic expressions. Substructures or properties of objects may be referenced in PASCAL-like fashion as

[1]This research was supported in part by NSF grant SED-7912803 in the Joint National Science Foundation - National Institute of Education Program of Research on Cognitive Processes and the Structure of Knowledge in Science and Mathematics, and in part by the Defense Advanced Research Projects Agency under contract MDA-903-80-c-007.

[2]which can be compiled to machine language by the Lisp compiler.

SE042635

" <record>:<field> ". Object-centered programming is supported for user-defined objects.

The GLISP language is easily extensible. Operator overloading for user-defined objects occurs automatically when arithmetic operators are defined as Messages for those objects. The compiler can compile optimized code for access to objects represented in user-specified *Representation Languages*, including database systems; GLISP has thus far been interfaced to the representation languages GIRL [7] and LOOPS [2]. GLISP has also been extended as a hardware description language for describing VLSI designs.

2. Object Descriptions

An Object Description describes the actual data structure occupied by the object; in addition, it describes Properties (values which are computed rather than being stored as data), Adjectives (which may be used in predicate expressions to test features of the object), and Messages to which the object can respond. The following example illustrates a GLISP object description. The name of the object type, CIRCLE, is followed by a description of the actual data structure occupied by the object: a Lisp list of the CENTER, which is of type VECTOR, and the RADIUS, which is a REAL number. The remaining items describe properties, adjectives, and messages for this object type.

```
(CIRCLE  (LIST (CENTER VECTOR) (RADIUS REAL))

  PROP   ((PI             (3.1415926))
          (AREA           (PI*RADIUS↑2))
          (DIAMETER       (RADIUS*2))
          (CIRCUMFERENCE (PI*DIAMETER)) )

  ADJ    ((BIG            (AREA>100)))

  MSG    ((DRAW           DRAWCIRCLEFN)
          (GROW           (AREA←+100))       )
```

As this example illustrates, the syntax of object descriptions makes it easy to define computed properties of objects.

3. Context and Type Inference

One of the design goals of GLISP is that program code should be independent of the implementation of the structures manipulated by the code to the greatest degree possible. Inclusion of redundant type declarations in program code would make the code dependent on the actual implementation of structures; instead, GLISP relies on type inference and its compile-time Context mechanism to determine the types of objects.

The Context is analogous to a symbol table, associating a set of named objects with their types. When a function is compiled, the Context is initialized to contain the function's arguments and their types; the other types used within the function can be derived by type inference. During compilation, the type of each intermediate expression is computed and propagated together with the code which computes the expression. The type of any substructure retrieved from a larger structure is retrieved by the compiler from the structure description of the larger structure. Assignment of a value to an untyped variable causes that variable to be assigned the type of the value assigned to it. Type inference is performed automatically by the compiler for the common "system" functions of Lisp. The type of the value computed by a user function may be declared; usually, the compiler is able to infer the type of the result of a function which is compiled, and it saves a declaration for the result type. Using these mechanisms, the compiler is able to perform type checking without requiring redundant type declarations within program code.

When Properties, Adjectives, and Messages are compiled, the compilation takes place within a Context containing the object whose properties are being compiled. Direct reference to the properties and substructures of the object is permitted within the code which defines properties; this is analogous to a ***with ... do*** in PASCAL. For example, the definition of `AREA` of a CIRCLE contains direct references to the stored value `RADIUS` and the property `PI`.

4. Recursive Compilation

The compilation of Properties, Adjectives, and Messages in GLISP is fully recursive. That is, when a property is to be compiled, the definition of the property is taken as a new expression to be compiled and is compiled recursively in the appropriate context. This allows an abstract datatype to define its properties in terms of the properties of its component abstract datatypes.

5. Compilation of Messages

Object-centered programming, which treats data objects as active entities which communicate by sending Messages, was introduced in SIMULA [1] and popularized by Smalltalk [4] [5]. In GLISP, the sending of a message to an object is specified in the form:

```
(SEND <object> <selector> <arguments>)
```

where the function name "`SEND`" specifies the sending of a message to `<object>`. The `<selector>` denotes the action to be performed by the message. When a message is executed at runtime, the `<selector>` is looked up for the type of the actual `<object>` to which the message is sent to get the name of the function which executes the message. This function is then called with the `<object>` and the actual `<arguments>` as its arguments. In effect, a message is a function call in which the dynamic `<object>` type and the `<selector>` together determine the function name. Interpretive lookup of messages is

computationally expensive -- often more than an order of magnitude more costly than direct execution of the same code. However, the types of objects can usually be known at compile time [3]. When the response to a message can be uniquely determined at compile time, GLISP compiles in-line code for the response; otherwise, the message is interpreted at runtime as usual. By performing message lookup only once at compile time rather than repeatedly during execution, performance is dramatically improved while retaining the flexibility of object-centered programming.

Associated with each Message selector[3] in an Object Description is a *Response* specification which tells how to compile the corresponding message; the Response contains code and a property list. There are three basic forms of Response code. In the first form, the Response code is simply a function name, and the code which is compiled is a call to that function. The second form allows the user to specify open compilation, in which the named function is "macro-expanded" in place with the actual argument values and types substituted for those of the function's formal arguments. The third form specifies a list of GLISP code, which is recursively compiled in place of the Message reference.

The latter form of Response code is a convenient and powerful way of defining computed properties of objects. The more usual way of defining such properties by means of small functions which compute them has several disadvantages. Function call overhead (and message-lookup overhead in an object-centered system) is expensive for small functions. Since function names must usually be unique, long function names proliferate. The syntactic overhead of writing small functions and calling them discourages their use. In GLISP, function call overhead is eliminated by expanding the Response code in place; the resulting code is then subject to standard compiler optimizations (e.g., constant folding). Names of properties do not have to be unique because they are referenced relative to a particular object type. Response code is easy to write and easy to reference.

6. Hierarchical Inheritance

Smalltalk and other object-centered languages organize objects into a hierarchy of classes and subclasses; this provides economy of representation by allowing features which apply to all members of a class to be described only once, at the class level. GLISP treats Object Descriptions as classes, and allows Properties, Adjectives, and Messages to be inherited from parent classes in the hierarchy. Response code which is inherited is compiled recursively *in the context of the original object*; this allows Response code in a class to be written in terms of features which are represented differently in its offspring classes.

---

[3] And likewise with each Property or Adjective: Property and Adjective references are compiled as if they were Message calls without any `<arguments>`.

An object can be a member of multiple hierarchies; this provides more power for representing "orthogonal" properties of objects. For example, a Planet object can be specified as being a PhysicalObject and a Sphere. If a program references the DENSITY of a Planet, the definition of DENSITY as MASS/VOLUME can be inherited from PhysicalObject and compiled recursively in the context of the original object, a Planet; the definition of VOLUME can then be inherited from Sphere and computed in terms of the Planet's RADIUS. In contrast, a different kind of PhysicalObject might have VOLUME stored directly as data but require that MASS be computed from WEIGHT. Thus, properties such as the definition of density or the volume of a sphere can be defined once, at a high level, and can then become effective for wide classes of objects.

7. Data Abstraction in GLISP

Data abstraction delays binding between the abstract features of objects and the implementation of those features until compile time [12]. By doing so, data abstraction can make program code smaller, less error-prone, and easier to change. In this section, abstraction features provided by GLISP are discussed.

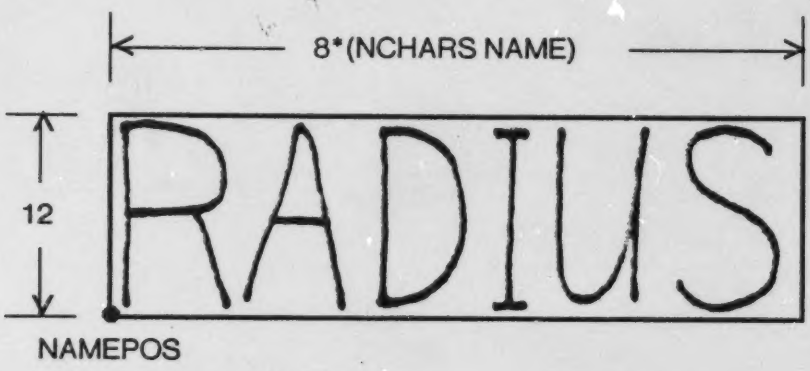
7.1. Independence of Form of Stored Data

Compile-time property inheritance allows objects which are implemented in different ways to share the same property definitions. For example, vectors might have X and Y values of various possible types (e.g., integer or real) stored in various ways. A single abstract class `VECTOR` can define vector properties (e.g., how to add vectors) which can be inherited by the various kinds of vector implementations.

A second form of structure independence is independence of the set of "equivalent" values which are stored. For example, a Circle implementation might equally well store either the radius or diameter of the circle. In GLISP, the syntax for referencing substructures and Properties of objects is the same, so the distinction between stored and computed values is hidden. A circle implementation in which diameter is stored can define `RADIUS` as a property, and can then inherit all the properties of the abstract type `CIRCLE` shown earlier. If code is to be independent of the set of properties which are actually stored, it must be possible to "store into" computed properties as if they were actually stored properties; GLISP permits a computed property which is an algebraic function of a single stored value to be "assigned a value" as well as being "read". For example, given an object C which is a CIRCLE, a program may include the code `(C:AREA ←+ 100)` ("the AREA of C is increased by 100"). The compiler will automatically compile code to compute the AREA from the stored RADIUS value, add 100 to it, compute the corresponding RADIUS for that AREA, and store the result back into the RADIUS datum.

7.2. Virtual Objects

In some cases, one would like to view an object as being an object of a different type, without creating a separate data structure for the alternate view. For example, a name which is drawn on a display screen might be viewed as a Region (a rectangle on the screen) for testing whether the display mouse is positioned on the name. Such a region is shown in the diagram below:



GLISP allows such a view to be specified as a *Virtual Object* , which is defined in terms of the original object. A virtual object definition for the name area illustrated above is:

```
(NAMEREGION  ((VIRTUAL REGION WITH
                   START = NAMEPOS ,
                   WIDTH = 8*(NCHARS NAME) ,
                   HEIGHT = 12)))
```

Given this definition, properties of the abstract datatype `REGION` can be used for the virtual object `NAMEREGION`; in particular, the message which tests whether a region contains a given point can be inherited to test whether the name region contains the mouse position.

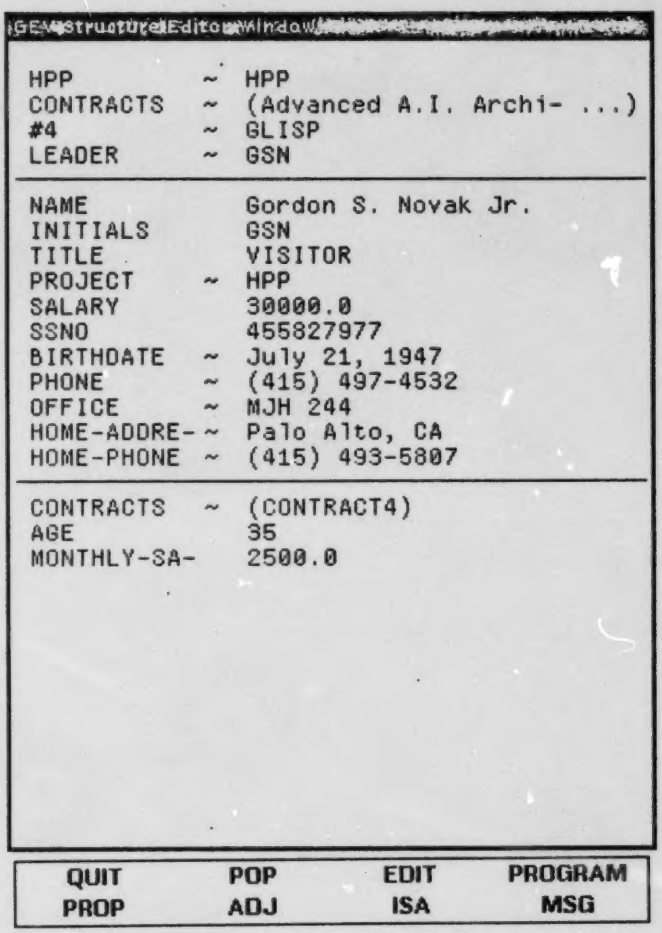
7.3. Compilation of Generic Functions

GLISP can compile a generic function for a specified set of argument types, resulting in a closed Lisp function specialized for the particular arguments. For example, given a generic function for searching a binary tree and a view of a sorted array as a tree, GLISP produces a binary search of a sorted array.

8. Interactive Features

GLISP provides an interactive programming environment which is complementary with the Lisp environment and provides support for abstract datatypes. Interactive versions of GLISP statements are provided for creating objects, sending messages to them, and retrieving their properties and substructures. Interfaces to the Lisp editor are provided for editing GLISP functions and aostract datatype descriptions.

GEV[4] [11] is an interactive display-based program which allows the user to inspect data, "zoom in" on features of interest, edit objects, display computed properties, send messages to objects, and interactively write programs. GEV is initiated by giving it a pointer to an object and the type of the object. Using the datatype description of the object, GEV interprets the data and displays it within a window, as shown below.



Edit Path

Stored Data

Computed Properties

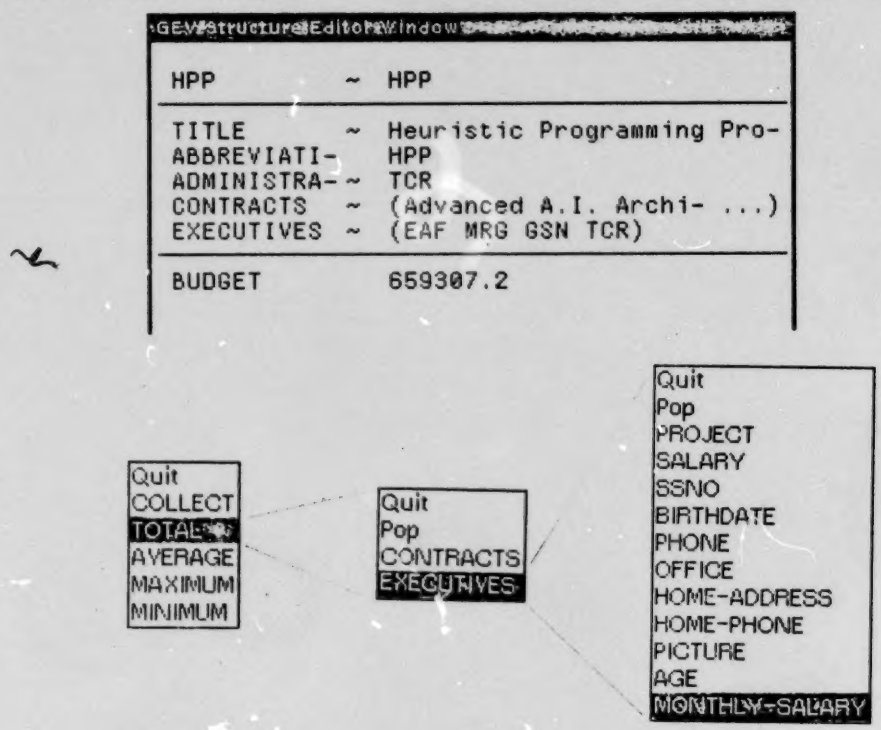
Data is displayed in the window in three sections: the Edit Path (which shows the path by which the currently displayed object was reached from the original object), the actual data contained in the object, and computed properties which have been requested or are specified to be displayed automatically. Often, the full value of

[4] For GLISP Edit Value.

an item cannot be displayed in the limited space available. In such cases, the SHORTVALUE property of the object is computed and displayed; a tilde (~) before the value indicates a shortvalue display.

Most interaction with GEV is done using the display mouse. If the name of a displayed item is selected, the type of that item is printed. If a value is selected, GEV performs a "push" to that value, displaying it in greater detail. The command menu below the display window is used to specify additional commands to GEV. The EDIT command calls a type-specific editor, or the Lisp editor, on the current object. The PROP, ADJ, and MSG commands cause a menu of the available Properties, Adjectives, or Messages for the type of the current object to be displayed; the property selected from this menu is computed for the current object and added to the display.

The PROGRAM command allows the user to interactively create looping programs which operate on the currently displayed object, using menu selection. This process and its result are illustrated below.



TOTAL MONTHLY-SALARY OF HPP EXECUTIVES = 15936.08

After the PROGRAM command is selected, a menu is presented for selection of the operation to be performed. The next menu specifies the set over which the program will operate; it contains all substructures of the current object which are represented as lists. Next, menus of all appropriate items are presented for

computed or stored data visible from the previous item (initially the current item in the loop) until a terminal item type (e.g., a number) is reached. GEV constructs from the program specifications a GLISP program to perform the specified computation, compiles it, and runs it on the current object. The results of the program are printed and added to the display.

The user of GEV does not need to know the actual implementations of the objects which are being examined. This makes GEV useful as an interactive database query language which is driven by the datatype descriptions of the objects being examined.

9. Discussion

We have discussed the methods by which the GLISP system provides several novel programming language capabilities:

1. An extended form of abstract datatype, including properties, adjectives, and messages, is provided. Datatypes may be organized in multiple hierarchies with inheritance of properties.

2. Optimized compilation is performed for object-centered programming by performing inheritance at compile time.

3. Generic programs and expressions can be compiled for a variety of specific datatypes.

4. Interactive programming and display-based editing are provided for abstract datatypes.

10. Implementation Status

GLISP and GEV, including all features described in this paper, are running and are being used for application programs at several sites. Versions of the compiler for the major dialects of Lisp are available on request.

11. Bibliography

1. Birtwistle, Dahl, Myhrhaug, and Nygaard. *SIMULA Begin.* Auerbach, Philadelphia, PA, 1973.

2. Bobrow, D.G. and Stefik, M. The LOOPS Manual. Tech. Rept. KB-VLSI-81-13, Xerox Palo Alto Research Center, 1981.

3. Borning, A. and Ingalls, D. A Type Declaration and Inference System for Smalltalk. Proc. 9th Conf. on Principles of Programming Languages, ACM, 1982.

4. Goldberg, A., et al. Special Issue on Smalltalk. In *BYTE Magazine*, , August, 1981.

5. Ingalls, D. The Smalltalk-76 Programming System: Design and Implementation. 5th ACM Symposium on Principles of Programming Languages, 1978.

6. Liskov, B., Snyder, A., Atkinson, R., and Schaffert, C. "Abstraction Mechanisms in CLU." *CACM 20*, 8 (Aug. 1977).

7. Novak, G. S. GIRL and GLISP: An Efficient Representation Language. Tech. Rept. TR-172, Computer Science Dept., Univ. of Texas at Austin, March, 1981.

8. Novak, G. S. GLISP: An Efficient, English-Like Programming Language. Proc. 3rd Annual Conf. of Cognitive Science Society, Univ. California, Berkeley, 1981.

9. Novak, G. S. GLISP Reference Manual. Tech. Rept. STAN-CS-82-895, Computer Science Dept., Stanford Univ., Jan., 1982.

10. Novak, G. S. GLISP: A High-Level Language for A.I. Programming. Proc. 2nd National Conference on Artificial Intelligence, Carnegie-Mellon University, Pittsburgh, PA, 1982.

11. Novak, G. S. The GEV Display Inspector/Editor. Tech. Rept. HPP-82-32, Heuristic Programming Project, Computer Science Dept., Stanford Univ., Nov., 1982.

12. Wiederhold, Gio. Binding in Information Processing. Tech. Rept. STAN-CS-81-851, Computer Science Dept., Stanford Univ., May, 1981.

13. Wulf, W.A., London, R., and Shaw, M. "An Introduction to the Construction and Verification of Alphard Programs." *IEEE Transactions on Software Engineering SE-2*, 4 (Dec. 1976).